Tools from ISO 19103

A GeoAPI Interface Proposal

Bryce Nordgren
USDA Forest Service

Legal

NOTICE OF RELEASE TO THE PUBLIC DOMAIN

This work was created by employees of the USDA Forest Service's Fire Science Lab on official time funded by public money. It is therefore ineligible for copyright under title 17, section 105 of the United States Code. You may treat it as you would treat any public domain work: it may be used, changed, copied, or redistributed, with or without permission of the authors, for free or for compensation. You may not claim exclusive ownership of this work because it is already owned by everyone. Use this documentation entirely at your own risk. No warranty of any kind is given.

Complete details are provided via the US Government's public websites:

- http://www.copyright.gov/title17/92chap1.html#105
- http://www.gpoaccess.gov/uscode/ (enter "17USC105" in the search box.)

Table of Contents

1 Introduction....1
2 Interpretation of 19100 UML diagrams....1
3 Primitive Data Types....2
3.1 Numeric Data Types....2
3.1.1 Discussion of unmapped items....3
3.1.1.1 Decimal....3
3.1.1.2 UnlimitedInteger....3
3.2 Text Data Types....4
3.2.1 Discussion of unmapped items....4
3.3 Date and Time....5
3.4 Truth....5
3.5 Multiplicities....6
3.5.1 UML Diagram....6
3.5.2 Multiplicity....6
3.5.2.1 Attribute: lower....7
3.5.2.2 Attribute: upper....7
4 Implementation and Collection Types....7
4.1 Collection Types....7
4.1.1 Templates, Generics, and Type Safety....7
4.1.2 Mapping....8
4.1.3 TransfiniteSet....8
4.1.4 Collection....8
4.1.5 Set....8
4.1.6 Bag....8
4.1.7 Sequence....9
4.1.8 Dictionary....9
4.2 Names and Namespaces....9
4.2.1 Namespace....11
4.2.1.1 Attribute: global....11
4.2.1.2 Attribute: acceptableClassList....11
4.2.1.3 Attribute: name....11
4.2.1.4 Role: names....11
4.2.1.5 Method: select()....11
4.2.1.6 Method: locate()....12
4.2.1.7 Method: generateID()....12
4.2.1.8 Method: registerID()....12
4.2.1.9 Method: unregisterID()....12
4.2.2 GenericName....12
4.2.2.1 Attribute: object....13
4.2.2.2 Role: scope....13
4.2.2.3 Method: parsedName()....13

4.2.2.4 Method: depth()....13
4.2.3 LocalName....13
4.2.3.1 Method: toString....13
4.2.3.2 Method: depth()....14
4.2.3.3 Method: parsedName()....14
4.2.4 ScopedName....14
4.2.4.1 Method: head()....14
4.2.4.2 Method: tail()....14
4.2.4.3 Attribute: object....15
4.2.4.4 Role: scope....15
4.2.4.5 Method: push()....15
4.2.4.6 Method: scopedName()....16
4.2.5 TypeName....16
4.2.5.1 Inherited Attribute: object....16
4.2.6 MemberName....16
4.2.6.1 Inherited Attribute: object....16
4.2.6.2 Attribute: attributeType....16
4.3 Records and Schemas....17
4.3.1 Inferences from UML definition....17
4.3.1.1 Inheritance....17
4.3.1.2 Type/Instance relationship....17
4.3.1.3 Named data access. 17
4.3.1.4 Unique member names....17
4.3.1.5 Self-Reference Relationships....17
4.3.1.6 NameSpace usage....19
4.3.2 Type....22
4.3.2.1 Attribute: typeName....23
4.3.2.2 Method: recordTypeRepresentation....23
4.3.2.3 Attribute: serializable....23
4.3.3 RecordType....23
4.3.3.1 Method: locate....23
4.3.3.2 Derived attribute: members....23
4.3.4 Schema....24
4.3.4.1 Method: locate....24
4.3.4.2 Method: asRecordSchema....24
4.3.4.3 Attribute: schemaName....24
4.3.5 RecordSchema....24
4.3.5.1 Inherited Method: asRecordSchema....24
4.3.5.2 Method: locate....24
4.3.6 Record....25
4.3.6.1 Role: recordType....25
4.3.6.2 Method: locate....25
5 Derived Data Types....25

5.1 Units....25
5.2 Measures (or Quantities)....25

1 Introduction

ISO 19103 defines many of the common tools and utility methods which are used as building blocks in other standards of the 19100 series. Some of the tools are basic, like numeric and boolean types. Others define standardized concepts, such as a Collections framework, or a system for specifying units and measures. These concepts are defined in a platform-neutral way, so that each platform may implement conceptually equivalent tools which differ only in implementation details from platform to platform. These tools may later be used to build more complicated constructs (like Application Schema) if the particular implementation details for the platform-at-hand are retained for reference.

GeoAPI, as an interface-only implementation of the 19100/OGC series of standards, specifies the form of Java-based geospatial code. GeoAPI, however, has tended toward specifying some of the more complex constructs without first outlining and defining the more basic constructs. Typically, when a basic construct is needed, the individual implementor is required to make a decision as to what correctly represents the form and function of the basic construct. This is a two-fold problem: in the first place, two implementors working separately could very well make different choices; and in the second place, 19103 is not a freely available standard so the implementor may be forced to guess at the functionality of a class based on the context in which it is used.

This document is intended to fix both problems. On the one hand, it should be used to specify a comprehensive, *single* mapping from the basic tools in 19103 to the Java language and the GeoAPI interfaces. On the other, where the precise functionality is not intuitively obvious to an experienced Java programmer from the class name alone, this document should take pains to describe the exact functionality desired. Lastly, where 19103 is not specific, this document should more precisely define the expected behavior and characteristics of the defined classes within the context of the Java/GeoAPI environment.

Precise definitions are the key to interoperability, especially if elements of a system are assembled from *different* libraries. If clients are not able to predict the behavior of an object, they will not be able to use it with predictable, repeatable results. However, precision should not go so far as to preclude implementation flexibility. Description of the desired behavior should stop short of prescribing a single means of accomplishing that behavior.

2 Interpretation of 19100 UML diagrams

This section contains a handful of notes designed to help potential implementors interpret UML diagrams which accompany specifications in the ISO 19100 series.

1. Any class which is stereotyped: <<Type>> is not intended to specify internal representation, regardless of what the UML diagram looks like.
2. Attributes in a class with a <<Type>> stereotype do *not* have to be directly implemented as attributes. If all of the public attributes found in the 19100 series of standards were literally implemented, encapsulation would be virtually nonexistent. Literal interpretation is not the intent of these standards.
3. The corollary to item 1 is that classes which do *not* have a <<Type>> stereotype *are* intended to specify internal structure, and hence literal implementation of attributes as indicated.
4. There are three groups of basic data types: Primitive types, Implementation and collection types, and Derived Types. These are the subject of the remainder of this paper, as mappings to the Java platform

are required for each and every one of them.

5. CodeLists are stereotyped <<CodeList>>, and are user-extensible enumerations. These are used when a collection of likely or common values are known, but other values are possible (and allowed). “CodeList can be used to describe an open enumeration. This means that it needs to be represented in such a way that it can be extended during system runtime.”
6. Enumerations are stereotyped <<Enumeration>>, and are not user-extensible. These are used when the complete set of values is defined by a standard, and all other values are disallowed.
7. Authors of 19100-series standards are supposed to avoid multiple inheritance!

3 Primitive Data Types

Primitive data types are “fundamental types for representing values”. These types include numeric representations, boolean values, strings, basic date and time objects, etc. Most of the types in this category have direct representations in many different implementation environments, including Java. This section, therefore, will primarily consist of a mapping from 19103 data type to Java data types, with an exposition of any caveats of which the user should be aware.

3.1 Numeric Data Types

The mapping for primitive data types is found in Table 1. Note that some items have more than one mapping. In these cases, it is intended that the implementor choose the mapping most appropriate to the task. Numeric types are mapped both to Java primitive numeric types and to the corresponding numeric objects.

Little or no discussion is required here, as most type mappings are obvious. ISO 19103 specifies that all the basic operations apply to these numeric data (add, subtract, multiply, divide, max, min, greater than, less than, equal to, casting, conversion to or from a string, etc.) The supplied Java mappings permit all of these operations, but there are of course differences between the primitive numbers and their Object counterparts. The concepts of “Integer” and “Real” do not carry with them an implied precision. They are meant to be mapped to any integer or floating point type which has an *appropriate* level of precision.

ISO 19103	Java
Decimal	None
Vector	something in vecmath...
Real	float, double, java.lang.Float, java.lang.Double
Number	java.lang.Number
UnlimitedInteger	org.opengis.util.UnlimitedInteger
Integer	byte, short, int, long, java.lang.Byte, java.lang.Short, java.lang.Integer java.lang.Long

Table 1: Mapping of numeric data types from 19103 to Java.

3.1.1 Discussion of unmapped items

As shown in Table 1, UnlimitedInteger and Decimal have no direct mapping to corresponding classes in Java.

3.1.1.1 Decimal

Decimal is a class which permits the exact representation of fractional quantities. This is a class capable of representing the quantity 0.1 with no error (as the fraction 1/10). There may also be a whole number component on the other side of the decimal place, as in 12.75. This item does not appear frequently in the derivative standards. It is different than Real in that Real contains inevitable floating point error.

POSTPONE: No implementation for the Decimal data type is proposed at this time.

3.1.1.2 UnlimitedInteger

An UnlimitedInteger is not equivilent to a java.math.BigInteger. BigInteger is an arbitrary-precision integer. An UnlimitedInteger is just an integer associated with an “infinite” flag. The numeric portion of the UnlimitedInteger retains the precision defined for it by Java. This data type is primarily used in the representation of Multiplicities, and hence sees fairly heavy usage. The proposed UnlimitedInteger interface is presented in Figure 1.

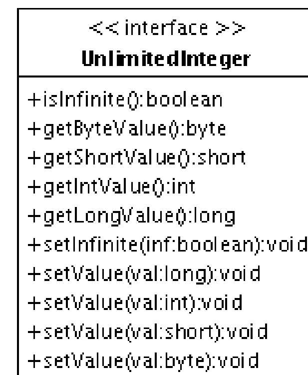

Figure 1: Specification of UnlimitedInteger

3.1.1.2.1 Attribute: infinite

The infinite attribute is mandatory, meaning that it always has a value. It is a flag which indicates whether the class represents an infinite value or not. If

this attribute is true, then the “value” attribute is meaningless and the results of the various get<Type>Value methods should be ignored.

There is no prescribed default value for this attribute.

3.1.1.2.2 Attribute: value

The value attribute is represented by four “getter” methods and four setter methods. Each of the getter methods conforming to the naming pattern: get<Type>Value() returns the numeric value cast to the appropriate type. The corresponding setter methods store a value of the indicated type.

The user is cautioned that this could result in loss of precision. The interface itself does not specify what variable is used to implement this attribute. This decision is left up to the implementor. The user must ensure that the implementation uses an integer with an appropriate amount of precision.

This attribute is optional. No value need be set. There is no default specified for the value of this attribute.

3.2 Text Data Types

The text data types in ISO 19103 are the minimum set of classes required to specify a String data type as a sequence of characters with a user specified encoding. The default character set encoding for the 19100 series of standards is ISO/IEC 10646 (Unicode). The only other predefined encoding is ISO 8859. The default encoding for java.lang.String is UTF-16, which is specified by Amendment 1 of ISO/IEC 10646. There are other variants of Unicode available, but this document will interpret all references to ISO 10646 to mean UTF-16, so that the default encodings become the same.

ISO 19103	***Java***
CharacterString	java.lang.String
Sequence<Character>	java.lang.CharSequence
Character	char
CharacterSetCode	None

Table 2: Mapping of text data types from 19103 to Java.

3.2.1 Discussion of unmapped items

The CharacterSetCode CodeList, has only two predefined values. Because it is a CodeList, other values are permissible. Java, on the other hand, does not have a class which represents a defined set of character encodings. One of the constructors to String takes a named encoding as a parameter. The defined name which corresponds to ISO 8859 is “ISO-8859-1”. The defined name to explicitly specify the default is “UTF-16”. Other defined encodings for the Java platform are: “US-ASCII”, “UTF-8”, “UTF-16BE”, “UTF-16LE”. (The “BE” and “LE” suffixes refer to “big endian” and “little endian”, respectively.)

3.3 Date and Time

The date and time types specified by ISO 19103 are extremely simple mechanisms for communicating century, year, month, day, hour, minute, and second. This capability is broken down into two classes. A “Time” type has a mandatory “hour” attribute, and optional “minute”, “second”, and “timeZone” attributes. Likewise, a Date contains a mandatory “century” and optional “year”, “month”, and “day” attributes. Finally, there is a DateTime type which inherits all the attributes from both Date and Time. All attributes in both of these classes are specified as character strings.

ISO 19103	*Java*
Date	java.util.Date
Time	java.util.Date
DateTime	java.util.Date

Table 3: Mapping of date and time types from 19103 to Java.

As shown in Table 3, all date and time types map to java.util.Date. However, this Java utility class contains a long integer which counts the milliseconds from an epoch in 1970, and contains no methods to set or retrieve centuries, years, or any of the other attributes of a 19103 date. The mapping in this case is imprecise. If the user needs access to the individual attributes of a time or date as Strings, then they may configure a java.text.DateFormat class to perform the storage or retrieval. Alternatively, they may use a java.util.GregorianCalendar to operate on specific fields.

The date and time types of ISO 19103 really map to a combination of java.util.Date and java.text.DateFormat or java.util.GregorianCalendar. The particular combination is determined by the needs of the user to access the data fields. However, the type required to transport basic time information is unquestionably java.util.Date. Attributes possessing any of these temporal types should use java.util.Date in their parameter lists or as their return types.

3.4 Truth

Truth in ISO 19103 is a much more extensively defined concept than is found in the typical programming language. There are three types of truth objects: Boolean, Logical, and Probability. As expected, the Boolean type has only two values: true or false. Logical adds a “maybe”. Probability allows any value between 0 and 1. As all of these are related concepts, numbers are assigned to the discrete values in Boolean and Logical. False is zero, true is one, and maybe is one-half.

Java only has a direct parallel for the “Boolean” data type.

ISO 19103	Java
Probability	None
Boolean	boolean, java.lang.Boolean
Logical	None

Table 4: Mapping of Truth data types from ISO 19103 to Java

As shown in Table 4, Boolean maps to either the boolean primitive type or to java.lang.Boolean, whichever is convenient. The other two classes have no direct mapping, and do not seem to be used often enough to merit their definition in GeoAPI.

3.5 Multiplicities

A multiplicity defines “the lower and upper bounds of the number of possible instances.” These data types have no existing counterpart in Java, and so they must be created.

ISO 19103	Java
Multiplicity	org.opengis.util.Multiplicity
MultiplicityRange	org.opengis.util.Multiplicity

Table 5: Mapping of multiplicity data types from ISO 19103 to Java

As shown in Table 5, the two ISO 19103 classes have effectively been merged into one. This was done because ISO 19103 defined a MultiplicityRange class with the same effective structure as “Multiplicity” in Figure 2. The class which ISO 19103 names “Multiplicity” has only one attribute, and that is the MultiplicityRange class. In essence, classes were defined where only one was needed. This proposal trims out the superfluous class.

3.5.1 UML Diagram

The UML diagram of the proposed multiplicity type is shown in Figure 2.

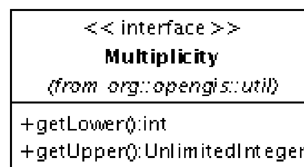

Figure 2: Proposed class to represent Multiplicity in GeoAPI

3.5.2 Multiplicity

This interface represents two read-only attributes: lower and upper. Both attributes are mandatory, and “upper”

must be greater than or equal to "lower." (If "upper" is infinite, it is greater than "lower.") This interface does not specify the type of the integer used to store the attributes, but data are transferred with "int" primitive data. Common values for "lower" would be zero or one, so "int" is perhaps overkill for this application.

3.5.2.1 Attribute: lower

The lower bound of the number of possible instances. This attribute is required.

3.5.2.2 Attribute: upper

The upper bound of the number of possible instances. If there is no upper limit, set the infinite attribute to true.

4 Implementation and Collection Types

This section contains the definition of three major components: a Collections framework, a system for Namespaces, and a system of Records and Schemas.

4.1 Collection Types

Collection types are basically an incarnation of the Java collections framework. The one major difference is that they all implement the TransfiniteSet interface.

4.1.1 Templates, Generics, and Type Safety

Every member of the 19103 collections framework is a "template class", meaning that they are declared with a syntax very similar to "generics" in Java (e.g., "Set<CharacterString>" would describe a set containing only strings.)

This is not the same as generics in Java. The Java notion of generics is a compile-time-only notion. There is only one "Set" interface in Java. The generics notation permits the compiler to perform additional type checking, adding a measure of safety to the code. At runtime, however, the compiler has stripped away the generics notation. True template programming (e.g., found in C++) actually generates, compiles, and links a separate class for every type requested. For more information, see Sun's website.

The challenge, then, is for implementors to ensure that their usage of Generics in Java is compatible with the notion of true template programming, because these particular concepts are defined in terms of templates, not generics. Additionally, for older JVMs (prior to 1.5), there is no notion of generics or template programming. In this situation, the implementor must take extreme caution to ensure that their use of collections "follows the rules." In this case, that means that all items in the collections framework contain only one specific, predefined type. If a collection contains strings, it may contain *only* strings. No collection may contain both strings and dates, or strings and numbers.

Obeying the rules of these collection types is completely up to client programmers, especially in the context of GeoAPI. When interfaces are compiled to java 1.4 compatibility, they lose whatever protection is granted them by generics. It is also possible to be a client running java 1.5, but using a GeoAPI implementation for 1.4. In this

situation, “generics” would seem to give the appearances of type-safety, but it is certainly possible that a bug in the implementation could return a mixed-type collection. As the type-safety check was performed at compile time, this mixed-type collection could be relayed to any java 1.5 based code which is expecting a type-safe collection.

4.1.2 Mapping

ISO 19103	***Java***
TransfiniteSet	org.opengis.spatialschema.geometry.TransfiniteSet
Collection	java.util.Collection
Set	java.util.Set
Bag	java.util.Collection
Sequence	java.util.List
CircularSequence	None
Dictionary	java.util.Map
KeyValuePair	java.util.Map.Entry

Table 6: Mapping of Collection types from ISO 19103 to Java.

4.1.3 TransfiniteSet

This interface is not explained *at all* in ISO 19103. It just appears in a UML diagram. Martin's comment in the javadocs: “A possibly infinite set; restricted only to values. For example, the integers and the real numbers are transfinite sets. This is actually the usual definition of set in mathematics, but programming languages restrict the term set to mean finite set.”

4.1.4 Collection

Collection is an empty, abstract class which implements the TransfiniteSet interface and serves as a parent to all collection types.

4.1.5 Set

A set is defined to be a finite collection of objects (in spite of the fact that it implements TransfiniteSet via inheritance from Collection.) It may not contain duplicate instances of an object. There is no specified order to the elements of a set.

4.1.6 Bag

A bag is a collection which may contain duplicate instances. There is no specified element order, as with a set.

The bag maintains an “occurence count” for each element instance, but ISO 19103 defines no attribute or method which advertises the number of items in the bag.

4.1.7 Sequence

A Sequence is a heavily used collection throughout the entire standard family. A Sequence is defined to be a Bag which orders the elements. Although not specifically stated, the ordering is strictly by index. This can be inferred from the statement that “a synonym to sequence is List”.

4.1.8 Dictionary

A dictionary is a mapping of key-value pairs. The conceptual model for a dictionary is an array where the indices are arbitrary objects instead of integers. This implies that the keys must be unique. ISO 19103 states that any objects are legal as either key or value, but typical usage in the standards will be to use a string as a key and numbers as values.

Template syntax in the definition of the Dictionary type requires that all keys have the same type and all values have the same type. The key type and value type may be different, however.

4.2 Names and Namespaces

The job of a “name” in the context of ISO 19103 is to associate that name with an Object. Examples given are objects: which form namespaces for their attributes, and Schema: which form namespaces for their components. A straightforward and natural use of the namespace structure defined in 19103 is the translation of given names into specific storage formats. XML has different naming rules than shapefiles, and both are different than NetCDF. This common framework can easily be harnessed to impose constraints specific to a particular application without requiring that a separate implementation of namespaces be provided for each format.

A construct known as QName is distributed with the JAXP reference implementation and is included with Java2 Standard Edition, version 1.5. This implementation represents more or less all of a fully qualified name with three “views” on that name: the local part, the entire URI, and the prefix. This is exactly what XML processors need, hence it's presence in the API for XML Processing.

However, this construct lacks the association with Object required to build constructs like records and schema (see section 4.3). The Eclipse Modeling Framework (EMF) contains a similar three-part notion of a namespace, but the names are constructed differently.

TODO: Investigate previous statement. If true, determine exactly how they are different. If false, eliminate it from the text.

Direct substitution of QName for namespace is not possible. The proposal in Figure 3 is advanced to capture the namespace functionality described in ISO 19103. There is, however, precious little documentation which accompanies the namespace UML diagram in the standard. Each class garners between one and three sentences, and individual methods receive no attention at all.

This document will provide considerably more detail, as precise definitions are required before use. Clients should be aware, however, that little or none of this information is present in the standard. As such, the bulk of

this section represents my interpretation of the UML diagram, and may at times be arbitrary. The extra information is advanced for the purpose of consistent interpretation of object behavior.

In GeoAPI 2.0, there is a partial implementation of the diagram in Figure 3. This has been based on an earlier version of the ISO 19103 standard. It is rumored that the previous version had an even more vague notion of a namespace than ISO 19103:2005(E) does.

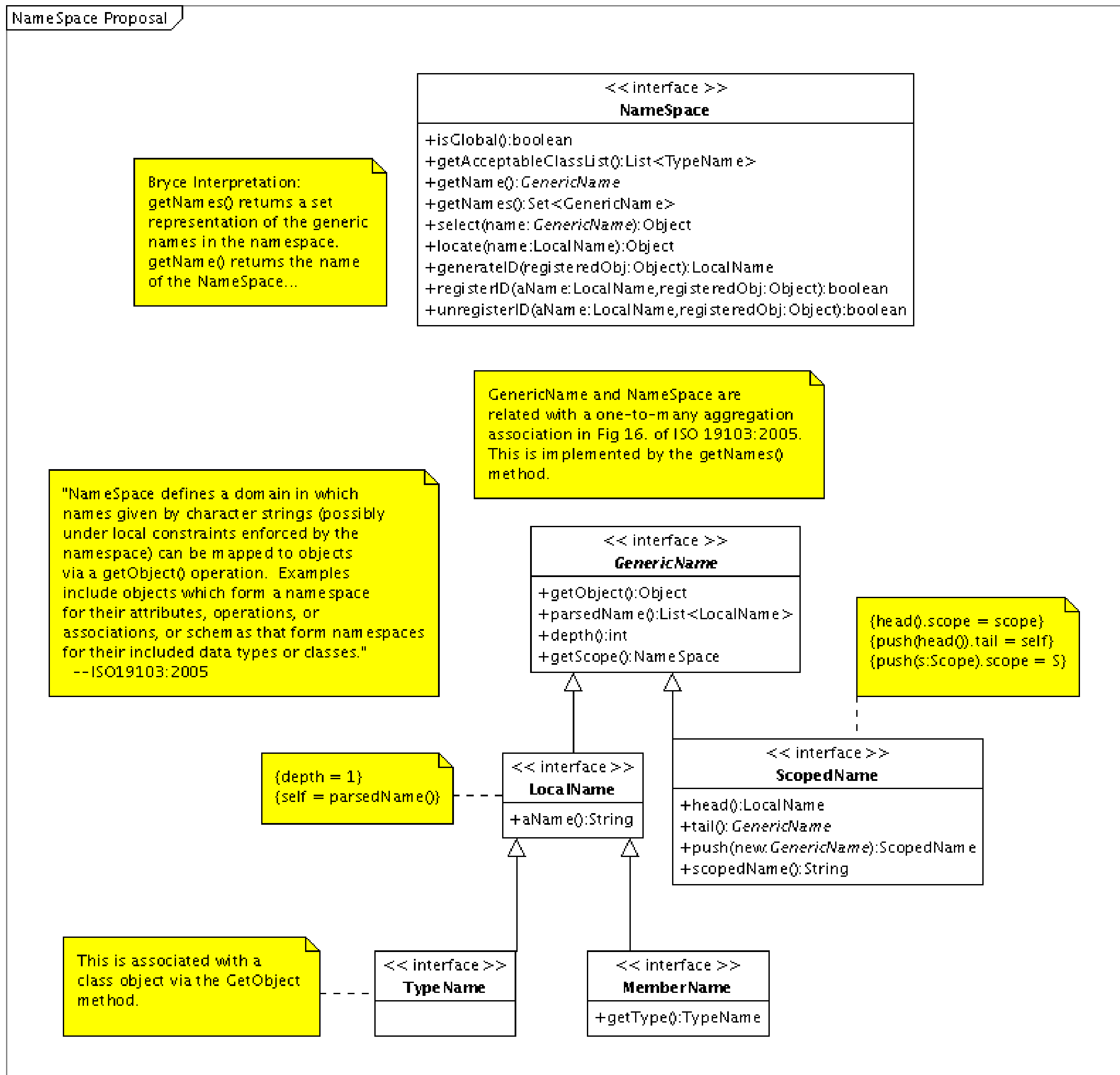

Figure 3: GeoAPI Namespace Proposal

4.2.1 Namespace

A namespace is an aggregation of 0 or more GenericNames via the "names" role. In Figure 3, this is accessed with the "getNames()" method. This was done to resolve a conflict with the name of the namespace, which was also represented with a method called "name".

The namespace contains utilities to look up any of the names it contains and return the object associated with that name. It contains utilities to "add" and "remove" names from the namespace. It also contains the ability to make up a name for an object when adding it to the namespace.

The following discussion of attributes and methods is completely fashioned by me and is open to interpretation.

4.2.1.1 Attribute: global

The "global" attribute indicates whether this namespace is a "top level" namespace. Global, or top-level namespaces are not contained within another namespace. There is no namespace called "global" or "root" which contains all of the top-level namespaces. Hence, this flag indicates whether the namespace has a parent.

4.2.1.2 Attribute: acceptableClassList

HELP: I really have no idea about this one!

4.2.1.3 Attribute: name

This attribute represents the identifier of this namespace. The object associated with this name should be the namespace itself.

If the global attribute is true, indicating that this is a top level NameSpace, then the name should be a LocalName. If false, name should be a fully-qualified ScopedName where head().scope.global == true.

4.2.1.4 Role: names

This is the set of GenericNames registered with this namespace. Because it is a Set, duplicate names are forbidden, as one would expect. Retrieve this set with the getNames() method in order to get a comprehensive list of all names within this namespace.

4.2.1.5 Method: select()

The select method returns the object associated with a GenericName. The GenericName must be valid in this NameSpace. For a LocalName, this means that the name must exist in the current NameSpace. The type of the object associated with this LocalName is unimportant. For a ScopedName, this means:

1. The head of a ScopedName must be a LocalName which is valid in this NameSpace. This LocalName must be associated with another NameSpace via the "object" attribute.
2. The tail must either be:
    1. a LocalName which is valid in the NameSpace associated with the head; or
    2. another ScopedName with these same constraints on head and tail, applied to the NameSpace

associated with the head.

If the GenericName is not valid in the NameSpace, this method throws java.lang.NoSuchFieldException.

4.2.1.6 Method: locate()

The locate method returns the object associated with a LocalName, if the LocalName is valid in this NameSpace. If the LocalName is not valid in the NameSpace, this method throws java.lang.NoSuchFieldException.

CONSIDER: Consider adding a variant of this method which accepts a string as the parameter to lookup.

4.2.1.7 Method: generateID()

This method takes an arbitrary object, fashions a name for it, constructs a LocalName to represent this name-object association, and registers the new LocalName with this NameSpace. For Java, the appropriate name to associate with an object is the final part of the fully qualified classname associated with the object. Special rules may be made for

4.2.1.8 Method: registerID()

This method adds the provided LocalName, object pair to this namespace. The provision of a registered object is actually redundant, as the LocalName is already associated with some object. The registered Object parameter is ignored and the LocalName is added to the namespace as-is. This method is the same as Set.add(). This method returns true if the addition succeeded and false if it failed.

CONSIDER: Is it rational to make the “aName” parameter into a string and have this method construct a LocalName to register in this namespace? This would then become a factory method for the creation of LocalNames.

4.2.1.9 Method: unregisterID()

As with registerID, the presence of the registeredObject parameter is superfluous. It is ignored. This method serves the same function as Set.remove().

CONSIDER: Is it rational to make the “aName” parameter into a string to allow users to remove LocalNames via their string representations?

4.2.2 GenericName

A GenericName is the top of the Name class tree. All Name objects share the characteristics defined here. Inspection of the GenericName interface reveals that all name objects:

1. have the ability to provide a “parsed” version of themselves;
2. are each associated with an object;
3. carry an indication of their “depth” (e.g., the number of hierarchical levels specified by the name); and
4. carry an association with a specific namespace, or scope, in which they are considered “Local”.

The same name object may not be registered in different namespaces. If the same character string is registered in two namespaces, two different name objects will result.

4.2.2.1 Attribute: object

This is the object with which the name is associated. It is accessed with the getObject() method. This attribute is mandatory. It may not be null.

CONSIDER: Martin thinks that making this attribute mandatory is wasteful in the context of CRS. Consider making this attribute optional for this object and mandatory for TypeNames and MemberNames.

4.2.2.2 Role: scope

This role maintains an association with a namespace in which this name is local. It is set on creation and is not modifiable. The scope of a name determines where a name “starts”. For instance, if a name has a depth of three (“coverage.ext.j2se”) and is associated with a NameSpace having the name: “org.geotools”, then the fully qualified name would be “org.geotools.coverage.ext.j2se.”

4.2.2.3 Method: parsedName()

This method returns a List of LocalNames which represent the global location of this name. All elements of the list except for the last one refer to Namespaces. The last element in the list refers to this object. The objects associated with LocalNames referring to namespaces are the namespaces to which they refer. The object attribute of the last item in this list is the object with which this name is associated.

4.2.2.4 Method: depth()

This method returns the depth of this name within the namespace hierarchy. This indicates the number of levels specified by this name. For any LocalName, it is always one. For a ScopedName it is some number greater than or equal to 2.

The depth is the length of the List returned by the parsedName method. As such it is a derived parameter.

4.2.3 LocalName

Local names are names which are directly accessible to and maintained by a NameSpace. Names are local to one and only one namespace. The namespace within which they are local is indicated by the inherited role: “scope”.

CONSIDER: It would perhaps be useful to create a method which returned the fully-qualified name as a string. This would involve concatenating the fully-qualified name of the associated NameSpace with this LocalName component.

4.2.3.1 Method: toString

This method returns the character string of the LocalName.

4.2.3.2 Method: depth()

The depth of a LocalName is always 1.

4.2.3.3 Method: parsedName()

The list returned by parsedName contains only one element: "self".

4.2.4 ScopedName

HELP: This section desperately needs a picture!

From ISO 19103: "ScopedName is a composite of a LocalName for locating another NameSpace and a GenericName valid in that NameSpace. ScopedName contains a LocalName as head and a GenericName, which might be a LocalName or a ScopedName, as tail."

From this brief definition (which is the entirety of the treatment of ScopedName in 19103), it may be seen that ScopedName is the means by which fully-qualified names are expressed. However, a ScopedName is not, in itself, what is commonly thought of as a "fully qualified" name. The ScopedName type is one link in the chain, not the entire chain. This interpretation is imposed by the fact that neither head nor tail is a list.

However, note that the definition of ScopedName allows for recursion. The tail may be either a LocalName or a ScopedName. If it is a ScopedName, then another link is forged—another step towards a remote LocalName is taken. If one looks at the ScopedName in the tail, *its* tail may be either a ScopedName or a LocalName. In this way, a ScopedName may represent an arbitrarily distant LocalName simply by the number of times the tail() evaluates to a ScopedName before finally terminating on a LocalName. The tail of a ScopedName, however, must terminate in a LocalName.

A ScopedName is essentially a stack element. There is no corresponding definition of a Stack data type because the *elements* contain a "push" method. This allows stack-like behavior as long as one only calls "push" on the ScopedName returned from the previous push.

4.2.4.1 Method: head()

The head is always a LocalName which has the same value for "scope" (e.g., is in the same NameSpace) as this ScopedName. Moreover, the definition from 19103 stated that this LocalName is to be used to locate another NameSpace. Combined, these two statements can be taken to mean that head() always returns a LocalName in the current namespace which refers to a child namespace.

The ScopedName is always specified relative to the NameSpace associated by the scope role. See rule #1 for determining the validity of a select() on a NameSpace in section 4.2.1.5. The ScopedName is a *fully-qualified* name if and only if head().scope.global is true.

4.2.4.2 Method: tail()

The tail may be either a LocalName or a ScopedName. According to 19103, the name in the tail should be valid in the NameSpace indicated by the head. See rule #2 for determining the validity of a select() on a NameSpace

in section 4.2.1.5.

If tail() returns a LocalName, the recursion is terminated because it has just returned the name we are trying to represent. If tail() returns a ScopedName, it means that the name we are trying to represent is “deeper” than this level and more intermediate NameSpaces must be traversed.

4.2.4.3 Attribute: object

The object associated with a ScopedName should be identical to the object associated with the LocalName which terminates the recursion in the tail.

4.2.4.4 Role: scope

The scope of a ScopedName should be identical to the scope of the head of the ScopedName.

4.2.4.5 Method: push()

The OCL for the push() method in the standard is nonsensical, as it refers to items which have not been defined. This section will attempt to establish a convention which makes sense.

According to the standard, this method creates and returns a new ScopedName object. The current ScopedName object (“self”, or “this”) is placed in the tail, and the provided GenericName must therefore be placed in the head.

There are two problems with this:

1. Head cannot be a GenericName, only a LocalName will do.
2. With this scheme, there is no way to terminate the tail recursion with a LocalName, as it is *always* populated with a ScopedName.

Simply inverting the problem will not do either. “Self” cannot be put into head because head must be a LocalName.

To solve this problem, we must consider the purpose of this method in the larger scheme of things. This method is a property of a stack *element* (not a stack) which attempts to create a stack. This process must begin by creating a ScopedName with a LocalName for both head and tail. The tail is the LocalName we are trying to represent, and the head is the name of the NameSpace within which this name lives. The first call to push should represent the name of the parent of this NameSpace, the next call would represent the grandparent, etc. This sequence of calls would provide only LocalNames to the push method.

However, the name of a NameSpace is fully qualified (see section 4.2.1.3), and push() accepts a GenericName as input. Rather than build up the ScopedName one hierarchical level at a time, the push() method could be called once, with the ScopedName of our name's enclosing NameSpace as an argument. If push() is supplied a ScopedName, it should be required to push the names of enclosing NameSpaces in order from deepest to shallowest.

The implication is that if push is provided a LocalName, it creates a ScopedName with “self” as the tail and the LocalName as the head. If push is provided a ScopedName, it iteratively calls push, supplying the elements of the parsedName() list in reverse order. (See section 4.2.2.3.) The first call to push is on “self”. The next call to

push is on the object returned from the first call to push(), etc.

Note that implementors must provide some external means to create the terminating ScopedName object (which possesses a LocalName in the tail.)

4.2.4.6 Method: scopedName()

This method returns the entire name as a string. There will be variants on this theme. XML afficianados may require URIs, for java classes, a dotted notation is more appropriate, for C++, a double-colon, for directories, a forward or reverse slash, and for CRS, it will depend on the mode of expression: URN or Authority:Identifier notation. Use an implementation suited to the needs at hand.

4.2.5 TypeName

This is a container for the name of a type. The inherited toString() method may be overridden to impose constraints on the name which are valid for the type being described.

4.2.5.1 Inherited Attribute: object

The object attribute is optional if the TypeName refers to a Type of unknown definition. The object attribute is mandatory if the TypeName refers to a RecordType (which implies that a definition will be supplied.) If provided, the object should be the NameSpace associated with this TypeName.

4.2.6 MemberName

This is a name designed to identify a member of a record. As such, this name bears an association with a type name. The inherited toString method may be overridden to impose constraints relevant to members of records. There may be alternate implementations of this: for instance, one implementation may apply to the in-memory model. Another may apply to a shapefile data store, etc.

4.2.6.1 Inherited Attribute: object

This optional attribute bears the same constraints outlined in section 4.2.5.1, applied to the TypeName “attributeType”. If the object is supplied, it must refer to a NameSpace, and that NameSpace must be initialized to have the same members as the RecordType specified in the “attributeType” attribute. This feature supports the limited association of RecordTypes as outlined in 4.3.1.5.1.

4.2.6.2 Attribute: attributeType

This specifies the type of the data associated with the record member.

NOTE: Figure 16 in 19103 specifies this attribute in the operations compartment of the class diagram.

4.3 Records and Schemas

Records and Schemas are similar to a "struct" in C, a table in SQL, a RECORD in Pascal, or an attribute-only class in Java if it were stripped of all notions of inheritance. They are organized into named collections called Schemas. Both records and schemas behave as dictionaries for their members and are similar to "packages" in Java.

Here is the complete treatment of Records, RecordTypes, and Schemas in ISO 19103: "A Record is a list of logically related elements as (name, value) pairs in a Dictionary. A Record may be used as an implementation representation for features." Everything else in this section is my opinion.

4.3.1 Inferences from UML definition

This section concentrates primarily on the development of inferences from the class diagram presented in Figure 5. These inferences serve to summarize some of the basic characteristics of schemas, records, and record types. This may be interpreted as an effort to intuit the guiding principles behind the class diagram. Clearly articulating the guiding principles should assist the effort to develop representative interfaces as well as compatible, consistent implementations.

4.3.1.1 Inheritance

The most obvious characteristic of records, schemas, and record types is that no notion of inheritance is present. All items are explicitly defined and contain only the members listed in the definition of that item.

4.3.1.2 Type/Instance relationship

Of the types defined in Figure 5, two are related by a data type to data instance relationship. A RecordType serves as the type definition to a Record, which acts as a data instance.

4.3.1.3 Named data access.

Schemas, RecordSchemas, RecordTypes, and Records all provide data access via name lookup. None of these objects possess a notion of an "order" to their members.

4.3.1.4 Unique member names

Schemas, RecordSchemas, RecordTypes, and Records are all specified to act as dictionaries for the members they contain. This requires that the member names be unique within the definition of a single type.

4.3.1.5 Self-Reference Relationships

The intent of the authors of ISO 19103 with respect to self-referential relationships is not clear from the class diagram. For example, it is difficult to determine from Figure 5 alone whether Schemas are allowed to contain other Schemas. To assist with the determination the relationships of identifier type to member value type are presented in Table 7. Schemas, RecordSchemas, and RecordTypes are treated individually in the text following

the table.

Class	*Identifier Type*	*Type of Member Value*
Schema	LocalName	Type
RecordSchema	LocalName	RecordType
RecordType	TypeName	TypeName
Type	TypeName	N/A

Table 7: Relationship of data types for identifiers and members of dictionary classes.

Another problem with self-referential relationships is that they mean different things to different classes. A RecordType is the definition of a data structure, therefore a RecordType which contains other RecordTypes implements an association between types. However, a Schema or RecordSchema exists to organize collections of Types or RecordTypes. A self-referential relationship with these organizational types should be taken to define a hierarchical namespace within which all the types exist.

4.3.1.5.1 RecordType

A RecordType is permitted to have limited association relationships with other RecordTypes. This is evident from the fact that a RecordType *is identified by*, and *has members of* the type: TypeName. A RecordType may therefore contain another RecordType as a member. This is called a limited association because a named member may be defined to be a single instance of some externally defined RecordType. This does not permit aggregation of any kind. Think of this as a composition with multiplicity one.

4.3.1.5.2 Schema

A Schema may not currently contain other Schemas as members. In the future, once some additional groundwork is laid either in the standards or by convention, Schemas may be allowed to contain other Schemas.

The roadmap to hierarchical Schemas is as follows: The contained Schemas must be identified by a TypeName. Additionally, a corresponding Type object must be constructed to refer to any Schema which is to be contained within another Schema. Finally, some means to permit Type objects to declare that it refers to a Schema type must be agreed upon.

A Schema is identified by a LocalName, and contains members of type “Type.” A “Type” object is little more than a TypeName used to *refer* to some external definition (e.g., int, java.lang.Class, “Coverage Core::CV_Coverage”, etc.) It is not the definition of the type itself. If a Schema which is to be contained in another is identified by a TypeName (which is a subclass of LocalName), a Type object could be constructed to refer to the containee. This Type object could be added to the container Schema.

Having done this, it is now appropriate to consider the use of such a construct. In particular, how is a member which happens to be a contained Schema to be differentiated from another member, which is not? If there is no mechanism by which a Type object which refers to a Schema is allowed to declare its special state, then it is impossible to distinguish a Schema from a RecordType (or an int, or a java.lang.Class.) This problem may be addressed in any number of ways: a naming convention may be adopted, whereby all Schema names conform

to a specific pattern; a namespace convention may be adopted, whereby all Schemas belong to a particular place in the namespace.

4.3.1.5.3 RecordSchema

A RecordSchema, similar to a Schema, has no current mechanism to support hierarchical definitions. As detailed in this section, it is possible that a RecordType could serve the same purpose. However, this requires that self-referential relationships for RecordTypes be interpreted as specifying a type hierarchy instead of specifying an association between RecordTypes. Like Schema, it is better to postpone the development of hierarchical frameworks with RecordSchema until some future date.

A RecordSchema is identified by a LocalName and contains members of type "RecordType". This situation is analgous to the relationship of Schemas to types, but is subtly different. For instance, a RecordType contains the definition of the type, whereas a Type contains only the type name. This difference leads directly to the expectation that all members of a RecordSchema contain named collections of types which can be accessed by name. Further, the expectation is that an instance can be made of any member of a RecordSchema. Possessing the definition of a type in addition to the name is a powerful, if subtle, difference from the situation of Schemas and their relation to Types.

A RecordSchema and a RecordType both act as dictionaries. A RecordType may have members of any Type. A RecordSchema, however, may only have members which are RecordTypes. Aside from the difference in class hierarchy, this is the sole functional difference between these classes. This leads directly to the observation that a RecordSchema is just a RecordType which only contains other RecordTypes as members. This leads to the observation that hierarchical RecordSchema may not be a worthwhile concept, as hierarchical RecordTypes are directly allowed by the UML diagram. This facility was noted in section 4.3.1.5.1.

4.3.1.6 NameSpace usage

Schemas, RecordSchemas, RecordTypes, and Records all define dictionaries which use some subclass of LocalName (see section 4.2.3) as the Dictionary's "key." All LocalNames have an associated NameSpace, in which they are considered to be "contained." There is also a parallel containment relationship in that each of the Dictionary's entries may be considered to be contained within the dictionary. These parallel relationships should be reflected in the usage of NameSpaces.

Parallel relationships between NameSpace and the various dictionary types is demonstrated for the case of the types java.lang.Double and java.lang.double in Figure 4. (The primitive type "double" is placed in the package java.lang for demonstrative purposes only.) Each large rectangle represents a dictionary, the type of which is specified by the column heading. In the upper-left corner of each dictionary rectangle is an identifier for the NameSpace associated with the dictionary. In the NameSpace column, this identifier is simply the name of the NameSpace. Subsequent sections detail how the other columns are associated with a particular NameSpace.

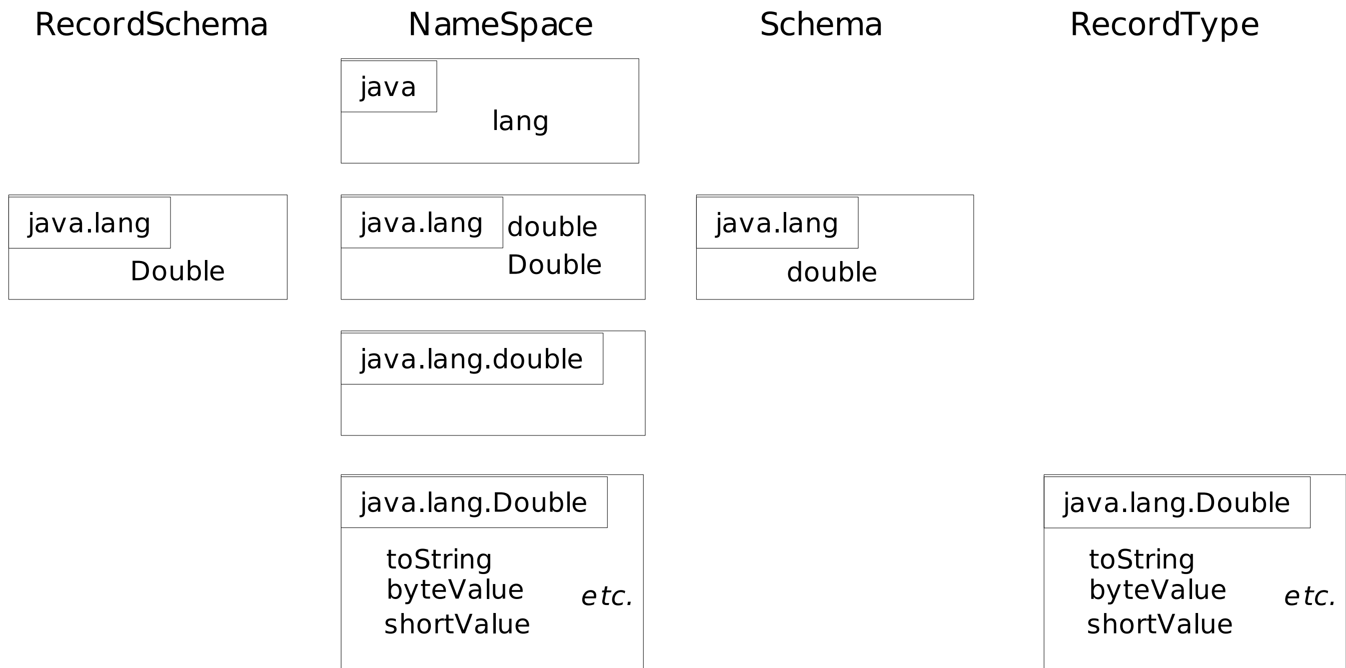

Figure 4: Illustration of the relationship between NameSpace and the various dictionary types.

Things to note about Figure 4:

1. A RecordSchema contains only those TypeNames for which an associated RecordType exists.
2. The type "java.lang.Double" would typically not be represented by a RecordType, as there would be no need to redefine it. It would certainly not contain names of *methods*. It was chosen because it is something which is not a primitive type. The reader is asked to suspend disbelief and assume that there is a RecordType named Double in the NameSpace java.lang with the attributes "toString", "byteValue", "shortValue," etc.
3. A NameSpace exists to contain every package (except for global packages), Type, RecordType, and all members of a RecordType.
4. The return values for members of the same name in different dictionary types is not necessarily the same. For instance: the name "Double" exists in the RecordSchema dictionary and the corresponding NameSpace. Calling locate on "Double" in the NameSpace returns the NameSpace java.lang.Double. When locate is called on "Double" in the RecordSchema, the associated RecordType is returned.

4.3.1.6.1 Hierarchical organizations of Schemas

As shown in section 4.3.1.5, neither Schemas nor RecordSchemas are currently capable of providing a hierarchical framework within which data types may be organized. NameSpaces, however, do define a hierarchical framework for arbitrary named items. Schema and RecordSchema participate in this framework by virtue of the fact that they are all identified by LocalName or some subclass. A Schema's location in the hierarchy is communicated by the name of the namespace which is associated to the "schemaName" property by the "scope" role. To express this another way: all names have an associated namespace, and all namespaces have a defined hierarchical position.

Individual Schemas and RecordSchemas must be considered to be somewhat less than “packages” in Java, as they are absolutely flat. They collect together Types and RecordTypes, respectively, of a single package but do not include subpackages. The organization of packages into a hierarchy is handled separately, via the construct of a NameSpace.

4.3.1.6.2 Scope of LocalNames for Dictionary Entries

The entries of dictionary types defined in Figure 5 are either TypeNames or MemberNames. Both types of names are subclasses of LocalName. LocalNames require a NameSpace in which they are considered local. This NameSpace shall be associated with the containing dictionary as specified in the following sections.

4.3.1.6.2.1 NameSpace of Schema and RecordSchema

Schemas and RecordSchemas are identified by the “schemaName” property, which is itself a LocalName. The scope role of the schemaName attribute shall be used to indicate hierarchy information, as described in section 4.3.1.6.1.

4.3.1.6.2.2 NameSpace of RecordType

A RecordType is different than a Schema or RecordSchema in that it does not carry its identifying name as a property. The identifier of a RecordType is maintained in the containing RecordSchema. It is the dictionary key which is associated with the RecordType. A RecordType, however, whether it is a member of a RecordSchema or a stand-alone type definition, must always have it's own NameSpace within which it may define a set of uniquely named fields.

In the simplest case, that of a stand-alone RecordType, there is no associated identifier and therefore no *imposed* NameSpace. RecordTypes in this situation exist outside any established type hierarchy. The MemberNames which define the fields of the RecordType, however, require a NameSpace for their scope role. Furthermore, this NameSpace must contain only the fields of this RecordType.

Handling RecordTypes within the type hierarchy (e.g., those with a containing RecordSchema) is more complex than handling a Schema or RecordSchema, as there are two modes of naming: definition and reference. Each RecordType has a single definition, but may be referenced in many places. These two cases are discussed separately.

The definition of a RecordType occurs when a RecordType is added to a RecordSchema, fixing its place in the hierarchy. This establishes a unique, fully-qualified name, maintained by the key of the RecordSchema, which may later be used to specify an association with another RecordType.

The process of defining an association with a previously defined RecordType may be described simply as adding the name of the previously defined type to the current record type. The name of the association role becomes the *key* in the current RecordType, and the fully-qualified TypeName of the previously defined RecordType becomes the associated *value*.

4.3.1.6.2.3 NameSpace of Instances (Records)

Records are intimately associated with RecordTypes, as the former instantiates the latter. Records, like

RecordTypes, carry no identification with them. Unlike a RecordType, they are not grouped into RecordSchemas, so they lack all association with a name. This is beneficial, as the inclusion of a name for a Record would require the generation of a unique name for every instance, which would soon prove unwieldy.

The NameSpace of a Record is not associated with the Record itself, but rather with the MemberNames it collects as dictionary keys. The fields of a Record are accessed by MemberName and all MemberNames require a valid "scope." The problem of defining a NameSpace for a Record, then, becomes the specification of a legitimate scope for all the MemberNames to belong to.

This problem is solved, quite simply, by requiring that the MemberName objects used as keys in a Record be identical to the MemberNames of the defining RecordType. Because they are identical, each field of a Record has the scope of the associated RecordType.

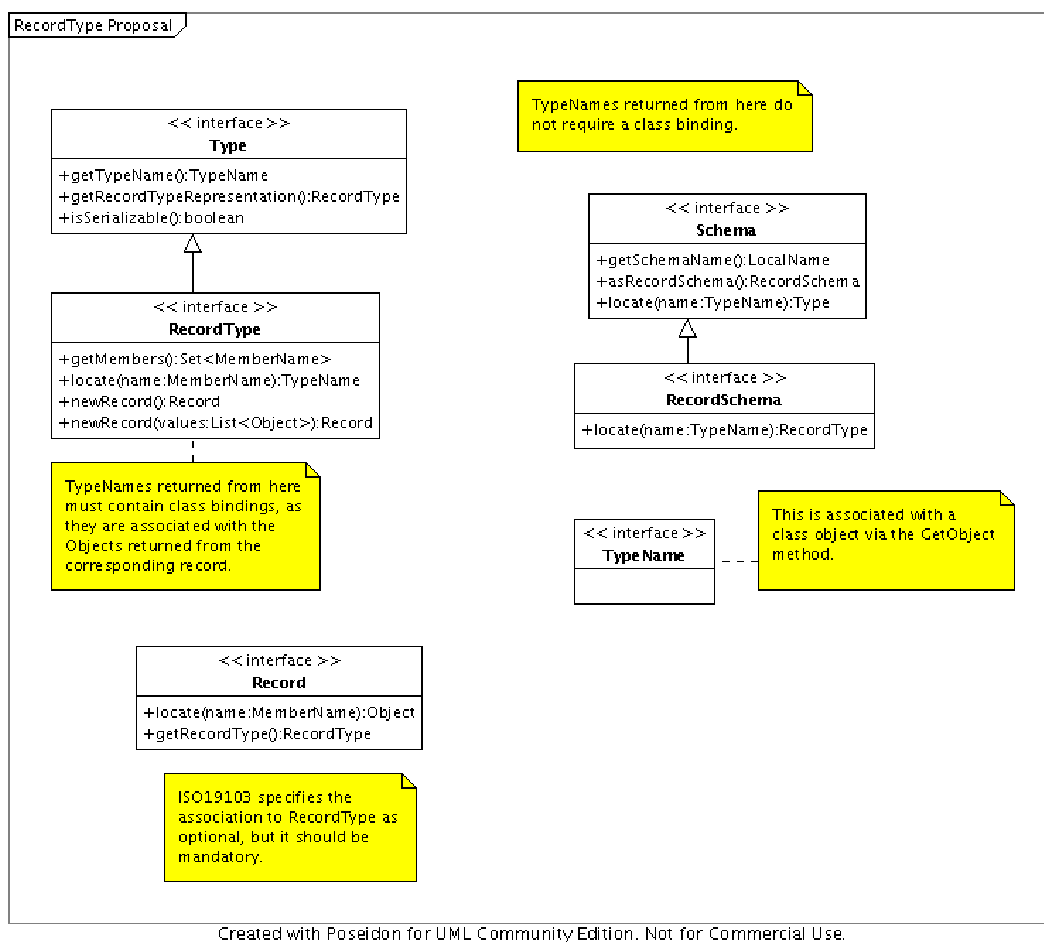

Figure 5: Proposal for Schema, RecordType, and Records

4.3.2 Type

A "type" is contained by Schemas and RecordTypes. In the context of a Schema, it is the value part of a

(TypeName, Value) pair in a dictionary. In the context of a RecordType, it is the value part of a (MemberName, Value) pair. A "Type" serves as a placeholder: it is a name for a predefined type where the definition of the type may or may not be known. A specialization of "Type" known as RecordType is the case where the definition is known.

4.3.2.1 Attribute: typeName

This is the name of the type. Note that this name has an associated scope and therefore this is a fully-qualified name.

4.3.2.2 Method: recordTypeRepresentation

If this object is a RecordType, this method performs the cast to RecordType and returns the result. If this object is not a RecordType, a ClassCastException is thrown.

4.3.2.3 Attribute: serializable

If the type is a Java class, this attribute is true if the class implements Serializable. If it is a RecordType, it returns true if all members of the RecordType are Serializable.

4.3.3 RecordType

A RecordType defines a flat, dynamically constructed data type. This object, once defined, may be used as a factory to create Records which are associated with this definition. A RecordType always has an associated NameSpace. If this RecordType is added to a RecordSchema, then the associated NameSpace is added to the NameSpace of the RecordSchema.

This class has methods for data access, but no methods to dynamically add members. This approach ensures that once a RecordType is constructed, it is immutable. It also implies the existance of a utility class used exclusively to build and return fully constructed RecordTypes.

The NameSpace associated with a RecordType contains only members of this RecordType. There is no potential for conflict with subpackages.

4.3.3.1 Method: locate

This method looks up the provided member name and returns the associated TypeName. If the member name is not defined in this RecordType, this method returns null in accordance with the definition of java.util.Map. The return value of this call should be identical to the attributeType property of the provided MemberName.

4.3.3.2 Derived attribute: members

The getMembers method returns the set of keys (of type MemberName) defined in this RecordType's dictionary. If there are no members, this method returns the empty set.

4.3.4 Schema

A Schema is a catalog of Types which may or may not possess a known definition. All Schemas possess an associated NameSpace within which the type names are defined. A Schema is a flat data structure, similar to a Java package except that it contains only type definitions and does not contain the names of subpackages. However, because the associated NameSpace contains both the type names and the subpackage names, a Type and a subpackage may not have identical names.

4.3.4.1 Method: locate

This method looks up the provided TypeName and returns the associated Type. If the TypeName is not defined within this Schema, this method returns null, in accordance with the definition of java.util.Map.

4.3.4.2 Method: asRecordSchema

If this object is an instance of RecordSchema, this method performs the cast and returns the result. If not, it throws a ClassCastException. Therefore, in the "Schema" class, this method should always throw a ClassCastException.

4.3.4.3 Attribute: schemaName

This is the name of the Schema. The scope attribute of the Schema name is associated with a NameSpace which fixes this Schema to a specific location in the type hierarchy.

4.3.5 RecordSchema

A RecordSchema is a specialization of Schema which contains only types defined in terms of RecordType objects. A RecordSchema is a flat structure bearing the same relationship to a java package as a Schema. The same potential conflict between type names and subpackage names applies. Also like a Schema, a RecordSchema derives its place in the package hierarchy from its association with the NameSpace in the scope of it's schemaName attribute.

4.3.5.1 Inherited Method: asRecordSchema

This method overrides the definition in Schema to return the current RecordSchema object.

4.3.5.2 Method: locate

This method returns the RecordType associated with the provided TypeName. If the provided TypeName does not exist (or if it refers to a subpackage), null is returned.

TODO: The definition of this method as shown in Figure 5 will cause problems with Java 1.4 due to lack of type-narrowing. Revisit and eliminate this problem.

4.3.6 Record

The Record class may or may not be associated with a defining RecordType. This is presumably because a Record contains a type and a value for each member without reference to an external RecordType. The member type is available as an attribute of the MemberName. The member value is available as the result of the locate method call.

TODO: Consider adding the equivalent of a "java.util.Map.put" method in order to set the member values. This method should never be capable of adding keys to the dictionary, but should only be able to replace the values of existing keys.

A record implementation should extend java.lang.Serializable.

4.3.6.1 Role: recordType

This association role connects the record with its RecordType definition. This association is specified to be optional.

4.3.6.2 Method: locate

This method returns the value associated with the provided MemberName. Null is returned if the provided MemberName does not exist in the Record.

5 Derived Data Types

5.1 Units

5.2 Measures (or Quantities)